5.2 Operations on Mutable Data Types

In the last section, we introduced the concept of mutable data types, and saw how we could mutate Python lists with the `list.append` method. In this section, we'll survey some of the other ways of mutating lists and other mutable Python data types. For a full reference of Python's mutating methods on these data types, please see Appendix A.2 Python Built-In Data Types Reference.

`list.append`, `list.insert`, and `list.extend`

In addition to `list.append`, there are two other ways of adding new items to a Python list. The first is `list.insert`, which takes a list, an *index* and an object, and inserts the object at the given index into the list at the given index.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.insert(strings, 2, 'hello')  # Insert 'hello' into strings at index
        2
>>> strings
['a', 'b', 'hello', 'c', 'd']
```

The second is `list.extend`, which takes two lists and adds all items from the second list at the end of the first list, as if `append` were called once per element of the second list.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.extend(strings, ['CSC110', 'CSC111'])
>>> strings
['a', 'b', 'c', 'd', 'CSC110', 'CSC111']
```

Assigning to a specific list index

There is one more way to put a value into a list: by overwriting the element stored at a specific index. Given a list `lst`, we've seen that we can access specific elements using indexing syntax `lst[0]`, `lst[1]`, `lst[2]`, etc. We can also use this kind of expression as the *left side* of an assignment statement to mutate the list by modifying a specific index.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> strings[2] = 'Hello'
>>> strings
['a', 'b', 'Hello', 'd']
```

Note that unlike `list.insert`, assigning to an index removes the element previously stored at that index from the list!

Mutating sets

Python `sets` are mutable. Because they are unordered, they are simpler than `lists`, and offer just two main mutating methods: `set.add` and `set.remove`, which (as you can probably guess) add and remove an element from a set, respectively.[1] We'll illustrate `set.add` by

[1] `list` also provides a few mutating methods that remove elements, though we did not cover them in this section.

showing how to re-implement our `squares` function from the previous section with `set` instead of `list`:

```
def squares(numbers: set[int]) -> set[int]:
    """Return a set containing the squares of all the given numbers.

    ...
    """
    squares_so_far = set()
    for n in numbers:
        set.add(squares_so_far, n * n)

    return squares_so_far
```

Note that `set.add` will only add the element if the set does not already contain it, as sets cannot contain duplicates. In addition, sets are unordered whereas `list.append` will add the element to the end of the sequence.

Mutating dictionaries

The most common ways for dictionaries to be mutated is by adding a new key-value pair, or changing the associated value for a key-value pair in the dictionary. This does not use a function call, but rather the same syntax as assigning by list index.

```
>>> items = {'a': 1, 'b': 2}
>>> items['c'] = 3
>>> items
{'a': 1, 'b': 2, 'c': 3}
```

The second assignment statement adds a new key-value pair to `items`, with the key being `'c'` and the items being `3`. In this case, the left-hand side of the assignment is not a variable but instead an expression representing a component of `items`, in this case the key `'c'` in the dictionary. When this assignment statement is evaluated, the right-hand side value `3` is stored in the dictionary items as the corresponding value for `'c'`.

Assignment statements in this form can also be used to mutate the dictionary by taking an existing key-value pair and replacing the value with a different one. Here's an example of that:

```
>>> items['a'] = 100
>>> items
{'a': 100, 'b': 2, 'c': 3}
```

Mutating data classes

Python data classes are mutable by default.[2] To illustrate this, we'll return to our `Person`

[2] Technically there is a way to create immutable data classes, but this is beyond the scope of this course.

class:

```
@dataclass
class Person:
    """A person with some basic demographic information.

    Representation Invariants:
      - self.age >= 0

    """
    given_name: str
    family_name: str
    age: int
    address: str
```

We mutate instances of data classes by modifying their attributes. We do this by assigning to their attributes directly, using *dot notation* on the left side of an assignment statement.

```
>>> p = Person('David', 'Liu', 100, '40 St. George Street')
>>> p.age = 200
>>> p
Person(given_name='David', family_name='Liu', age=200, address='40 St. George
         Street')
```

One note of caution here: as you start mutating data class instances, you must always remember to respect the representation invariants associated with that data class. For example, setting `p.age = -1` would violate the `Person` representation invariant. To protect against this, `python_ta` checks representation invariants whenever you assign to attributes of data classes, as long as the `python_ta.contracts.check_all_contracts` function has been called in your file.